Memory and Copying

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

Objectives

By the end of this lecture, you should

- Be able to explain how large data structures in a functional language differ from similar structures in an imperative environment
- know how to simulate updates in a persistent environment
- be able to add undo functionality using persistent data

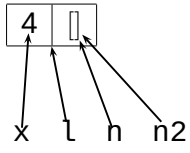
Type Constructors and Memory

- When a type constructor is invoked, it causes memory to be allocated.
  - Writing an integer. . .
  - Writing an empty list. . .
  - Using the cons operator :: . . .
- Writing down a variable does not cause memory to be allocated.

```
1 # let x = 4;;  allocates 4
2 # let n = [];;  allocates empty list
3 # let n2 = n;;  does NOT allocate memory
4 # let l = x::n;; A cons cell is allocated,
5   but not the 4 or the empty list
```


Memory Diagram

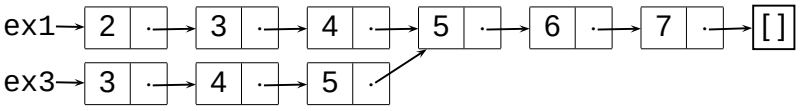

```
1 # let x = 4;;  allocates 4
2 # let n = [];;  allocates empty list
3 # let n2 = n;;  does NOT allocate memory
4 # let l = x::n;; A cons cell is allocated,
5   but not the 4 or the empty list
```



The Nature of Data in a Functional Language

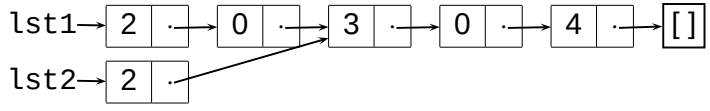
- In a functional language, data is not changed, it is *copied*.

```
# let incthree lst =
    match lst with
      a::b::c::xs -> a+1 :: b+1 :: c+1 :: xs;;
val incthree : int list -> int list = <fun>
# let ex1 = [2;3;4;5;6;7];;
val ex1 : int list = [2; 3; 4; 5; 6; 7]
# let ex2 = incthree ex1;;
val ex2 : int list = [3; 4; 5; 5; 6; 7]
```



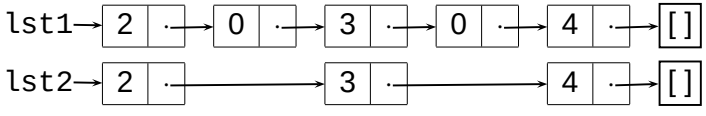
Delete First Zero

```
# let lst1 = [2;0;3;0;4];;
val lst1 : int list = [2; 0; 3; 0; 4]
# let rec delzero lst =
    match lst with
      0::xs -> xs
    | x::xs -> x :: delzero xs
    | [] -> [];;
# let lst2 = delzero lst1;;
val lst2 : int list = [2; 3; 0; 4]
```



What if I wanted to delete *all* the zeros?

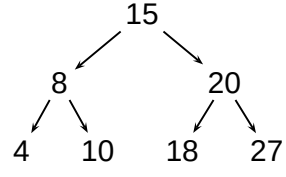
```
# let lst1 = [2;0;3;0;4];;
val lst1 : int list = [2; 0; 3; 0; 4];;
# let rec delzero lst =
    match lst with
      0::xs -> delzero xs
    | x::xs -> x :: delzero xs
    | [] -> [];;
# let lst2 = delzero lst1;;
val lst2 : int list = [2;3;4];;
```



Trees and Persistent Structures

Definition of Binary Search Tree

```
# type tree = Branch of int * tree * tree
            | Leaf of int | Empty ;;
# let t1 = Branch(15, Branch(8, Leaf 4, Leaf 10),
                      Branch(20, Leaf 18, Leaf 27));;
```



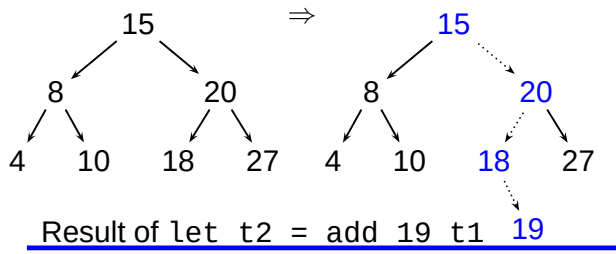
Suppose I want to add 19 to this tree?
(Note we are using a shorthand notation. . .)

add in scheme

```
1 (define (add tree item)
2   (if (eq? tree 'empty) (make-leaf item)
3     (if (leaf? tree)
4       (if (> (leaf-item tree) item)
5         (make-branch (leaf-item tree)
6                      (make-leaf item) 'empty)
7         (make-branch (leaf-item tree)
8                      'empty (make-leaf item)))
9       (if (> (branch-item tree) item)
10        (make-branch (branch-item tree)
11                     (add (branch-leftt tree) item)
12                     (branch-right tree))
13        (make-branch (branch-item tree)
14                     (branch-left tree)
15                     (add (branch-right tree) item)))
```


add

```
1 # let rec add item atree = match atree with
2     Empty -> Leaf item
3   | Leaf old -> if item<old then Branch(old,Leaf item,Empty)
4                             else Branch(old,Empty,Leaf item)
5   | Branch(old,t1,t2) ->
6       if item < old then Branch(old, add item t1, t2)
7                     else Branch(old, t1, add item t2)
```

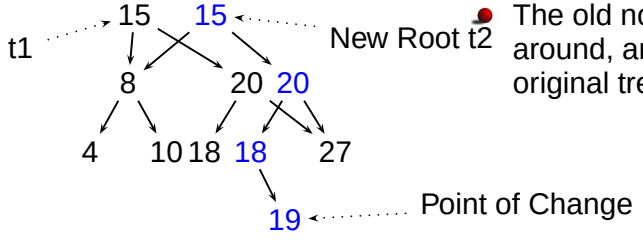


The black nodes are shared, the blue nodes are new.

Result of `let t2 = add 19 t1`

Adding to a tree, part 2

- When a change is made to data, new nodes must be created from the point of change to the root of the data.



- The old node `t1` is still around, and points to the original tree (without the 19).
- Suppose we now add 9 to this tree, and assign it to `t3`? (By the way, this is a trick question.)

How can we use this?

- At this point we can write an interpreter for a small language with "undo" capability.
- Commands: `Add`, `Show`, `Undo`.
- What will we need?
  - A way to process commands. We can use a data-type:

```
1 type command = Add of int | Show | Undo
```

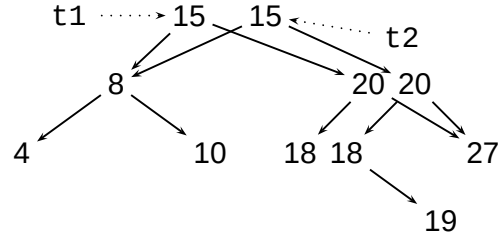
  - A notion of the *current tree*
  - A stack of *past trees*

Current Tree Command List Tree Stack

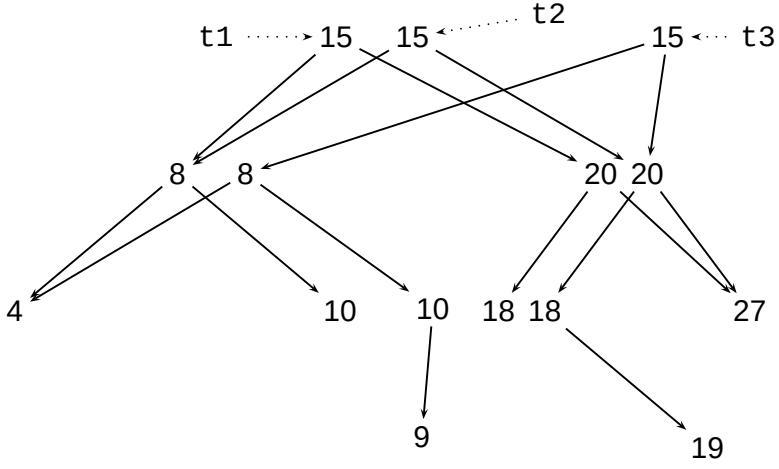
The Interpreter

```
1  let eval commands =
2   let rec aux tree commands stack =
3    match commands with
4      (Add i)::xs ->
5              aux (add i tree) xs (tree::stack)
6    | Show :: xs ->
7              show tree ; aux tree xs stack
8    | Undo :: xs ->
9              aux (List.hd stack) xs (List.tl stack)
10   | [] -> print_string "Done\n" in
11   aux Empty commands []
```


1. In the lecture we discussed the consequences of adding a second new element to the tree. Draw the result of the command `(define t3 (add t2 9))`.



2. Write the `delzeros` function, but make it able to share the data.




```
1  let rec getLastZero lst pos save = match lst with
2    | [] -> save
3    | 0::xs -> getLastZero xs (1+pos) (1+pos)
4    | x::xs -> getLastZero xs (1+pos) save
5
6  let rec delzero lst = match lst with
7    | 0::xs -> delzero xs
8    | x::xs -> x :: delzero xs
9    | [] -> [];;
10
11 let rec shareDelZero lst =
12   let last = getLastZero lst 0 0 in
13     (delzero (take last lst)) @ (drop last lst);;
```

If you like this, you will definitely want to see the work of Chris Okasaki. For example:

- Simple and Efficient Purely Functional Queues and Deques
- Catenable Double-Ended Queues
- Three algorithms on Braun trees